

RIFERIMENTO ORA CHE SORE (risparmio) ~ 10 SEZIONI CRITICHE

Si ha la **MUTUA ESCLUSIone** quando non più di un processo alla volta può eseguire le operazioni sulle variabili comuni.

Le operazioni sulle variabili comuni vengono eseguite.

La **SEZIONE CRITICA** è la sequenza d'intervalli con le quali un processo esegue o modifica un insieme di variabili comuni.

Lo **SCHEMA GENERALE** prende l'uso di un **INDICATORE** associato ad ogni classe di sezioni critiche, può essere **LIBERO** (quando nessuna sezione critica è ~~è~~ in esecuzione) oppure **OCCUPATO**.

Agni processo che vuole utilizzare una sezione critica delle dove esegue uno protocollo che inserisce l'accesso alla sezione critica tra **PROLOGO** ed **EPILOGO**

- 6 requisiti
- [1] Le sezioni critiche devono essere eseguite con le interruzioni abilitate **esclusiva**
 - [2] Le sezioni critiche di una classe devono essere eseguite in forma **mutualistica** (se: altri processi)
 - [3] se un processo si blocca all'esterno di una sez. critica non deve influenzare V
 - [4] La soluzione al problema di mutua esclusione deve essere ereditante (CONDIZIONALI)
 - [5] La soluzione non deve tenere il BUSY WAITING
 - [6] La soluzione non deve disabilitare per rimuovere un altro processo (STARVATION)
- * G = NO DEADLOCK

Nell'ipotesi di SEZIONI CRITICHE BREVI è possibile escludere i criteri 1,5

L'algoritmo di Dijkstra prende l'utilizzo di variabili baseline CIBERO1, LIBERO2 per indicare se un processo è in versione critica e un'unica variabile TURNO per indicare l'ID del processo a cui tocca eseguire la versione critica.

	PROCESSO P	PROCESSO Q
PROLOGO	<pre> CIBERO1:=true while LIBERO2 do: if TURNO==2 then begin CIBERO1:=false repeat until TURNO1; CIBERO1:=true end <A> TURNO:=2 CIBERO1:=false </pre>	<pre> LIBERO2:=true while LIBERO1 do: if TURNO==1 then begin CIBERO2:=FALSE repeat until TURNO=2 CIBERO2:=TRUE end TURNO:=1 CIBERO2:=FALSE </pre>
SEZIONE CRITICA		
EPILOGO		

Venne alternato il turno tra 2 processi, quando un processo uscì dall'area in ver. critica impostò CIBERO = false e impose il turno all'altro processo. Il ciclo while è un ciclo di this ottimo in cui il processo controlla continuamente il flag dell'altro processo ad il turno corrente → ESTENSIONE con N processi comporta complicazioni.

PRO: soddisfa requisti 1, 2, 3, 4 e 6

CONTRO: problema Breathing waiting, non scalabile per N processi.

L'ALGORITMO DI PETERSON usa le variabili LIBERO1, LIBERO2 per indicare se è in
versione critica e un'unica variabile ~~LIBERO~~ TURNO che indica l'id del processo e un
tocco registrare la versione critica

	PROCESSO P	PROCESSO Q
<u>PROLOGO</u>	LIBERO1 = true TURNO = 2 while LIBERO2 and TURNO ≠ 2 do;	LIBERO2 = true TURNO = 1 while LIBERO1 e TURNO ≠ 2 do;
<u>SEZIONE CRITICA</u>	<A>	
<u>EPICOLOGO</u>	LIBERO1 = false	LIBERO2 = false;

→ quando un processo vuole accedere alla sua versione critica impone il proprio flag
a "true" e il turno all'altro processo. Successivamente il processo entra in un ciclo
while in fase perché l'altro processo ha impostato il proprio flag a "true" e il turno
è ancora dell'altro processo.

PRO: evita deadlock (1), ovvero starvation (6), versioni critiche indipendenti (3)

due processi non possono entrare nella versione critica simultaneamente (2)

CONSO: presente problema del busy waiting ed è poco robusto al caso di N processi

(**ALGORITMO DEL RONDO**) è un algoritmo di mutua esclusione con N-macini (con due waiting), il problema viene modellato come il servizio dei clienti in un negozio: al momento delle richieste di servizio alle ~~risorse~~ ^{risorse} critiche, ciascun macino riceve un numero e viene servito il processo che prende il numero più basso.

Un macino generico P_i che ha intenzione di raggiungere una risorsa critica, attende cioè un altro macino della stessa classe che sta raggiungendo il numero opposto e presenta un processo con un numero assegnato più basso. \Rightarrow VABBENE PER LA COORDINAZIONE DISTRIBUITA, ovvero se il processo sa cosa hanno fatto gli altri.

VARIABILI CONDIVISE: var CHOOSING: array [0..N-1] di boolean "false".
NUMBER: array [0..N-1] di interi inizializzati a 0.

PROCESSO GENERICO P_i :

MACINO: CHOOSING[i] = true;
NUMBER[i] = 1 + max [NUMBER[0], ..., NUMBER[N-1]].

1) acquisizione numero d'ordine
for j := 0 to N-1 do
 CHOOSING[i] = false;

2) ritorno {
 while !CHOOSING[i] do;
 while NUMBER[j] \neq i and NUMBER[i,j] < (NUMBER[i], i) do;

} $j \neq i$
AND
 $(number[i]) = number[i]$
 $number[j] < number[i]$ or
 $number[i,j] < (Number[i], i)$

SEZIONE CRITICA <A>

EPICOGO NUMBER[i] = 0

PER: Soddisfa requisiti 1, 2, 3, C, G ed è SCALABILE. L'algoritmo può essere utilizzato in interni distribuiti in cui ogni macino dispone di sua memoria locale, ~~sentendo controllo della memoria~~.

CONTRO: presenta Breathing Waiting ed il numero di processi deve essere molto limitato. Ogni macino deve conoscere e coordinarsi reciprocamente.

* L'avventuale arresto di un nodo non blocca gli altri nodi; FAULT TOLERANCE.

2. SEMAFORI

Un **SEMAFORO** è una variabile mem negativa $s \geq 0$ con valore iniziale $s_0 \geq 0$. Ad ogni semaforo è associata una **LISTA DI ATTESA** Q_s , nella quale sono presenti i descrittori dei processi che attendono ed eseguire l'autorizzazione.

Sul semaforo sono ammesso solo due operazioni indirizzabili:

- **WAIT(,)** che può essere PASSANTE ($s > 0$) o Rientrante ($s = 0$); contest switch
- **SIGNAL(,)** sempre PASSANTE, non comporta alcuna modifica allo stato del processo che l'ha eseguito.

La reella del processo rispetta ordine d'arrivo tramite politica FIFO, inoltre le due operazioni vengono eseguite in modello Kernel \Rightarrow è protetta.

Un **SEMAFORO** è MURX se la variabile semaforo è ammessa solo i valori 0 ed 1

WAIT(,):
begin
 if $s = 0$ then
 < stato processo modificato in waiting, descrittore inserito in Q_s >
 else $s = s - 1$;
 end;

SIGNAL(,):
begin
 if $\exists k \in Q_s$ un processo in Q_s then
 < descrittore rimosso da Q_s ; stato del processo modificato in ready>
 else $s = s + 1$;
 end;

Ad ogni blocco di sezione critica viene associata una variabile semaforo, con valore 1 e
prologo ed epilogo vengono realizzati mediante `wait(n)` o `signal(s)`.

Solo il 1° processo avrà una `wait` nonante mentre i successivi avranno una `wait` bloccante.
^{PRIMITIVE SEMAFORICHE}
L'INDIVISIBILITÀ DEGLI SPATTI ~~SEZIONI CRITICHE~~ si tiene intendo le interruzioni per
la protezione dell'esecuzione sullo stesso processo e utilizzando il meccanismo `lock(x)`
ed `unlock(x)` per la protezione dell'esecuzione su processori diversi. \Rightarrow `WAIT(n)` e `SIGNAL(s)`
visti come sezioni critiche brevi.

→ `WAIT(n)` begin
< ~~per~~ diminuzione delle interruzioni >
 `lock(x);`
 ~~* codice della wait~~ * `unlock(x)`
 < ~~diminuzione interruzioni~~ >
end;

`SIGNAL(s)` ..
 `lock(x)`
 < codice della signal >
 `unlock(x)`
 ..

ESEMPI SEMAFORI

● PRODUTTORE-CONSUMATORE: Buffer limitato in grado di contenere N messaggi e cui accedono il produttore P in scrittura ed il consumatore C in lettura

VINCOLI: \rightarrow il produttore non può inserire un messaggio nel buffer se è pieno
 \hookrightarrow un consumatore non può prelevare un messaggio dal buffer se questo è vuoto.

↳ SOLUZIONE MEDIANTE 2 SEMAFORI, men-disp con $n_0=0$ per garantire la disponibilità di almeno un messaggio e spazio-disp con $n_0=N$ per garantire la presenza di almeno uno spazio vuoto ~~nel buffer~~

PROCESSO P

repeat

<modifica messaggio>

wait (spazio-disp)

<deposito messaggio>

signal (messaggio-disponibile)

forever

PROCESSO C

repeat

exit (men-disp)

<prelievo messaggio>

signal (spazio-disp)

<consumazione messaggio>

forever

↳ SOLUZIONE SIMMETRICA, inoltre P e C possono operare in parallelo sul buffer solo su messaggi diversi.

● PRODUTTORE-CONSUMATORI: generalizzazione del problema precedente. Ora l'azione di deposito è una versione critica per produttori, l'azione di prelevamento è una versione critica per i consumatori \rightarrow SI AGGIUNGONO ALTRI 2 SEMAFORI: mutex1 e mutex2 entrambi con $n_0=1$

P

(repeat)

<prod. mess>

exit (spazio-disp)

exit (mutex1)

<dep. mess>

signal (mutex1)

forever signal (men-disp)

C (repeat)

exit (men-disp)

exit (mutex2)

<prelievo mess>

signal (mutex2)

forever signal (spazio-disp)

<consumazione mess>

MUTUA ESCLUSIONE CON RISORSE EQUIVALENTI NO

↳ n risorse $\{R_1, \dots, R_n\}$ tra loro equivalenti ed m processi $\{P_1, \dots, P_m\}$ che devono operare in modo esclusivo su una qualsiasi risorsa R_j mediante una tra le operazioni $\{A, B, \dots\}$ disponibili per ogni risorsa

↳ SOLUZIONE: introduzione di una nuova risorsa G, gestore di $\{R_1, \dots, R_n\}$, che si occupa di mantenere una struttura dati contenente lo stato delle risorse $\{R_i\}$
operando due procedure $\begin{cases} \text{RICHIEDA}(x: 1..n) \\ \text{RILASCIO}(x: 1..n) \end{cases}$

- indurre sono richiesti:
- SEMAFORO MUX con $S_0 = 1$, per garantire la mutua esclusione di richiesta e di rilascio
 - SEMAFORO RIS con $S_0 = n$, poche indica il numero di risorse disponibili
 - Vettore di booleani $CIBERO[i]$ che memorizza quali risorse sono libere in ciascun intorno e quali sono occupate

Vor mux: numero = 1

Vor ris: numero = n

Cibero: array [1..n] bool = true

procedura richiesta ($x: 1..n$) procedura rilascio ($x: 1..n$) traccia esecuzione di P_i:

Vari: 0..n;

begin

 wait (ris);

 wait (mux);

 i := 0

 repeat i := i + 1;

 until (cibero[i]);

 x := i

 cibero[i] := false

 signal (mux);

end

begin

 wait (mux)

 cibero[x] := true;

 signal (mux);

 signal (ris);

end;

; traccia esecuzione di P_i:

; Vor S; 1..n

; Richiesta();

; uno delle risorse x-sima

; Rilascio(S);

2-BIS. SEMAFORI PRIVATI

I semafori privati sono usati su problemi di sincronizzazione in cui lo sforzo del programmatore da ridurre più ampiamente sulla base di CONDIZIONI DI PRIORITÀ dei processi.

Si definisce **SEMAFORO PRIVATO** P_i un semaforo privi con valore iniziale non zero.

Tale che:

1) p_i è inizializzato a 0

2) solo P_i può eseguire la $\text{wait}(P_i)$ su p_i

3) anche altri processi possono eseguire la $\text{signal}(P_i)$ su p_i .

sono usati per la REALIZZAZIONE DI POLITICHE DI GESTIONE DECCE RISORSE

(composte da)

• ACQUISIZIONE DECA RISORSA DA PARTE DI UN PROCESSO P_i

$\text{Wait}(\text{mutex})$

< verifica delle condizioni di sincronizzazione con esecuzione condizionata dello signal(p_i) >

$\text{signal}(\text{mutex})$

$\text{Wait}(\text{priv}_i)$

< uso della risorsa >

P_i analizza le variabili come nelle sezioni critiche protette

nel semaforo mutex e ne deduce che la propria condizione di sincronizzazione è verificata; modifica lo stato

delle variabili e inoltre all'acquisizione delle risorse tramite uno signal(p_i) rendendo l'ultima wait(priv_i) nonbloccante

RILASCO DI UNA RISORSA DA PARTE DI UN PROCESSO P_j

$\text{wait}(\text{mutex})$

< verifica delle condizioni di sincronizzazione con esecuzione condizionata dello signal(p_i) >

$\text{signal}(\text{mutex})$

può scegliere quale processo ridurre tra i processi in condizione di sincronizzazione

Alcune CONSIDERAZIONI sull'uso dei semafori privati.

- Ogni processo che durante l'acquisizione trova la risorsa non disponibile deve lasciare traccia in modo esplicito della propria占有zione entro la SEZIONE CRITICA
- Il processo che rilascia la risorsa deve eseguire signal (priv.) solo se c'è almeno un altro processo.
- La fase di acquisizione della ~~risorsa~~ deve essere segnata dalla fase d'uso, occorre lasciare traccia in modo esplicito all'interno della sezione critica delle non disponibilità delle risorse per gli altri processi.

Di seguito alcuni ESEMPI

• PRODUTTORI-CONSUMATORI CON MESSAGGI A CONGEZZA VARIABILE

usa: int risponi \rightarrow numero di processi PRODUTTORE risponi

int vuote \rightarrow numero di celle vuote del buffer

int richiesta[i] \rightarrow numero di celle richieste del produttore Pi risponi.

PRODUTTORE \rightarrow procedura acquisizione
Wait (mutex);

if risponi == 0 and vuote >= 1, then

begin \Rightarrow condizione incrociata
vuote--; \exists istante
signal (priv);
end

else

begin

risponi++;

richiesta[i]=1;

end

signal (mutex)

wait (priv);

wait (mutex);

CONSUMATORI \rightarrow procedura rilascio (m: integer)

Wait (mutex)

vuote = vuote + m

exit = false

while risponi < 0 and not exit {

\leftarrow individuazione tra i processi risponi
 del processo Pk con lo stesso richiesto, max

 il max vuote {

 vuote = vuote - max;

 richiesta[K] = 0

 - risponi = risponi - 1

 signal (priv); };

 else Exit = true };

 signal (mutex)

\leftarrow deposito nel buffer;

 signal (mutex1)

 { if Exit = true }

 signal (mutex)

• MUTUA ESCLUSIONE CON RISORSE EQUIVALENTI E PRIORITÀ TRA PROCESSI

Varibili: int nopen \Rightarrow numero di processi produttori nonari

NC

bool PS[i] \Rightarrow indica se il processo Pi è nonario

bool R[i] \Rightarrow indica se la risorsa i è occupata

int RA[i] \Rightarrow indice della risorsa assegnata al processo Pi

int disponibile \Rightarrow numero risorse disponibili

ACQUISIZIONE DELLA RISORSA

Wait (mutex)

if disponibile > 0 then

{ seleziona una risorsa K tra

quelle disponibili utilizzando il vettore R > quelle risorse utilizzando il vettore PS

disponibile--;

R[K]=1;

RA[i]=K;

signal (priv);

};

else {

nopen++;

PS[i]=1;

}

signal (mutex)

Wait (priv);

< uno delle risorse RA[i]

RILASCIO DELLA RISORSA

; Wait (mutex)

; if nopen <= 0 then

{

< seleziona il processo Pi a priorità minima tra

quegli nonari utilizzando il vettore PS > quegli nonari utilizzando il vettore PS

PS[i]=0;

R[i]=0;

nopen--;

} signal (priv);

else {

REm]=0

disponibile++;

};

signal (mutex);

PROBLEMA FILOSOFI

var state: array [0..N-1] tra (thinking, hungry, eating) \rightsquigarrow init. thinking

mutex: semaphore \rightsquigarrow S₀=1

priv: array [0..N-1] remolari \rightsquigarrow S₀=0

- procedure entry pickup () {

wait (mutex);

state[i] = hungry;

if (state[(i+N-1) mod N] <= eating) && (state[(i+1) mod N] <= eating) {

state[i] = eating;

signal (priv[i]);

}

signal (mutex);

wait (priv[i]);

}

- procedure entry putdown () {

wait (mutex);

state[i] = thinking;

if (state[(i+N-1) mod N] == hungry) && (state[(i+N-2) mod N] <= eating) {

state[(i+N-1) mod N] = eating;

signal (priv[(i+N-1) mod N]);

if (state[(i+1) mod N] == hungry) && (state[(i+2) mod N] <= eating) {

state[(i+1) mod N] = eating;

signal (priv[(i+1) mod N]);

signal (mutex);

→ evita deadlock (maneggiando stat.) ma non

(l'informazione è i-1 ed i+1 si alternano a turni)

i filosofi i scambiano sempre remolari

3. BLOCCO CRITICO ~ DEADLOCK

Se un thread in Thread non cambia più il suo stato, cioè se le risorse richieste sono ~~troppo~~ bloccate da altri thread in Thread, si ha una situazione di **DEADLOCK**.

Per blocco critico si intende una situazione nella quale uno o più processi o threads rimangono indefinitivamente bloccati a causa dell'impossibilità del verificarsi delle condizioni necessarie per il loro proseguimento.

Si distinguono:

- **risorse riutilizzabili**: redigo l'uso di un processo nonano avere risultato (in modo estremo)
- **risorse condivisibili**: segnali o messaggi scambiati tra i processi, possono esistere non oppure acquisiti da un processo.

Dati n processi $\{P_1, P_2, \dots, P_n\}$ ed m tipi di risorse $\{R_1, R_2, \dots, R_m\}$ si può verificare un deadlock solo se sono VERE CONTEMPORANEAMENTE le **condizioni di Havender**

- 1) Le risorse nonano essere utilizzate da un solo processo allo stesso tempo.
- 2) I processi ~~che si appoggiano~~ che si appoggiano alle risorse che già possiedono mentre richiedono risorse addizionali.
- 3) Le risorse già consegnate ai processi non nonano essere raffidate ad essi. (ASSENZA DI PREEMPTION)
- 4) Esistono almeno due processi $\{P_i, P_{i+1}, \dots, P_k\}$ tali che P_i è in Thread di una risorsa presieduta da P_{i+1} , che a sua volta detiene una risorsa presieduta da P_{i+2} etc.

↳ CAFENA CIRCOLARE DI PROCESSI IN ATTESA

(Le condizioni sono necessarie, dicono sufficienti SE si ha una sola istanza per ciascun tipo di risorsa.)

Gli approcci per il trattamento dello deadlock sono: DEADLOCK PREVENTION (prevenzione), DEADLOCK AVOIDANCE (prevenzione), DEADLOCK DETECTION AND RECOVERY (recovery).

DEADLOCK PREVENTION \rightarrow negare una delle 4 condizioni necessarie

1) GARANTIRE LA MUTUA ESCLUSIONE

2) ~~•~~ la condizione di non-estesa può essere negata in due modi:

- Richiesto in blocco di tutto l'insieme necessarie

- Richiesta di nuove risorse solo dopo il rilascio di quelle già non esistono

3) L'ASSENZA DI PREEMPTION può essere negata in due modi:

- rilascio delle risorse in corso di una richiesta di una nuova non immediatamente ^{dimentica}
- attribuzione della nuova richiesta ad un eventuale processo bloccato che la detiene.

4) ATTESA CIRCOLARE: si impone un ordinamento (inizio globale delle risorse, ad ogni tiro viene assegnato un numero intero che permette di confrontare due risorse e stabilire la precedenza nell'ordinamento definito).

DEADLOCK AVOIDANCE \rightarrow si verifica sulle liste dello stato corrente dell'allocazione

delle risorse e delle richieste complesse dei processi.

■ SEQUENZA SAVVY La sequenza di processi $\langle P_1, \dots, P_n \rangle$ è una sequenza salve per lo stato di allocazione attuale se, per ogni P_i , le richieste che P_i può ancora effettuare non sono entro raggiungibile utilizzando le risorse attualmente disponibili in aggiunta alle risorse non esistute dai processi P_j per i quali $j < i$.

La zona di impossibilità è l'insieme di stati di assegnazione delle risorse per i quali c'è almeno una risorsa assegnata a più processi, ovvero degli stati incompatibili.

Se lo stato corrente di un thread entra in una zona non salve \rightarrow si ha deadlock

■ ALGORITMO DEC BANCHIERE \rightarrow

DEADLOCK DETECTION AND RECOVERY

Si esamina ad intervalli prefissati o in circostanze specifiche \Rightarrow 3 una condizione di deadlock.

• ALGORITMO DI RIDUZIONE DEL GRAFO DI ALLOCAZIONE DELLE RISORSE

\hookrightarrow si eliminano dal grafo tutti i nodi la cui richiesta non sono soddisfatte, liberando risorse per altri processi, che, una volta soddisfatti, vengono eliminati.

Se nel grafo finale non restano più nodi di assegnazione o richiesta \Rightarrow il grafo completamente ridotto. In questo caso non sono presenti situazioni di deadlock.

• RECUPERO DA SITUAZIONI DI DEADLOCK: Terminazioni di processi (o di quelli in weak lock)

O di un processo alla volta fino ad eliminazione delle deadlocks; rotazione delle sinergie di alcuni processi e assegnamento ad altri fino ad interrompere la deadlock (problemi di selezione di una vittima, rollback e l'orrendum).

3B. REGIONI CRITICHE

• LETTORI-SCRITTORI CON SEMAFORI (priorità di lettura)

reader
writer
readprcount++;
signal(mutex)
<lettura>
...
wait(mutex)
readercount--
if readercount == 0 then
signal(w);
signal(mutex)

Scrittore
wait(w)
...
<scrittura>
...
signal(w)

• reader preference: lettore entra anche quando

è presente una scrittore in coda quando un lettore esce

* **STRONG**: viene nella coda lettore non esce

WEAK: viene nella coda lettore o uno scrittore non esce

WEAK-WEAK \Rightarrow viene nella coda scrittore se presente

• writer preference: scrittore del reader preferisce

Il contratto di regione critica semplice (RCS) consente di rendere unicamente gli oggetti di mutua esclusione.

di mutua esclusione

vor V: shared T;

Region V do COMPILATORE S;
end

Var mutex: semaphore initial(1);
Wait(mutex);
S;
signal(mutex);

Sia la sezione critica, gli statement hanno accesso alla variabile condivisa V ed il compilatore può verificare che la variabile V venga usata esclusivamente all'interno della regione critica e può implementarne correttamente la mutua esclusione, inoltre può riconoscere situazioni di deadlock potenziale.

La regione critica condizionale (RCC) ha un funzionamento simile a quello delle regione critica semplice, in aggiunta è presente ~~una~~ la possibilità di specificare una condizione di accesso alla regione critica B. Se B è vero, la regione critica è completata eseguendo i statement al processo libera la regione critica e si risponde in Q;

vor V: shared T

region V when B do:

S
end;

COMPILATORE

mutex semaphore = 1
signal semaphore = 0
count: int = 0

Wait(mutex);
while not B do
{

count++;

signal(mutex); \rightarrow libera ner. crit.

Wait(mutex); \rightarrow riserva in coda Q;

Wait(mutex);

end;

}

S;

while count != 0 do signal(mutex);
{ signal(mutex);
count--;

PRO: maggior diversità nel programma ed estensione del problema nel programma, controllo a tempo di compilazione

CONTRO: il programmatore non può controllare l'ordine con cui i processi hanno avuto il risorse comune

- (ecc) permette di operare sincronizzazioni solo all'inizio delle sezioni critiche
- Semplifica problemi di sincronizzazione
- In presenza di più processi nella coda Qv, tutti vengono rimigliati e provveduto a valutare la propria condizione B → può determinare un numero elevato di context-switch

ESEMPI

→ LETTORI-SCRITTORI

vor rw-buffer shared record
num lettore: integer 0
num scrittore: integer 0
occupato: boolean false
end

processo al buffer non avviene ~~mai~~ entro le regioni critiche perché
altrimenti si perderebbe parallelismo

traccia
num lettore
num scrittore

R
...
inizio lettura()
< Read >
fine lettura()

W
...
inizio scrittura()
< Write >
fine scrittura()

procedure inizio lettura();

begin
region rw-buffer () do

begin
avait (num scrittore=0);
num lettore++;

end

procedure fine lettura()

begin

region rw-buffer do begin
end

procedure initio-writter();

begin
region rw-buff do begin

num-writer++;
await ((nd occupied) and (num-left=0));
occupied=true;

end

end

procedure fine-writter()

begin
region rw-buffer() do

begin

num-writer--;
occupied=false;

end

end

FILOSOFI A CENA

for philosophers: shared record

state: array [0...N-1] com (thinking, eating) initial thinking

procedure pickup (i: 0..N-1)

begin

region philosophers when ($\text{note}[(i+N-1) \bmod N] < \text{eating}$ and $\text{note}[(i+1) \bmod N] < \text{eating}$)

do note[i]:=eating

procedure putdown (i: 0..N-1):

begin

region philosophers when true do note[i]:=thinking;

do end

with i filosofi numeri.

Le monitor

Un **MONITOR** è un oggetto astratto condizionato da una struttura dati, delle procedure, uno o più elementi iniziali (intuttori) ed una specifica di sincronizzazione. Le intenze di un monitor vengono trattate come variabili condizionate, per questo motivo le PROCEDURE ENTRY devono essere eseguite in mutua esclusione.

Lo SCOPO del monitor è quello di controllare l'eseguzione di routine tra problemi concorrenti in base a determinate politiche di gestione.

Avviene su 2 livelli:

- 1) garantisce che un solo processo alla volta abbia accesso al monitor unico semaforo mutex con valore iniziale pari ad 1. A questo punto i processi hanno accesso alla risorsa.
- 2) Controllo dell'ordine con cui i processi hanno accesso alla risorsa. Attraverso l'introduzione delle **VARIABILI DI TIPO CONDIZIONE**.

Le **VARIABILI DI TIPO CONDIZIONE** rappresentano una coda nelle quali sono inseriti i processi. Vengono implementate mediante un campo valore ed un campo puntatore. Funzioni analoghe a quelle dei semafori previsti.

- **COND_WAIT**: sospende SEMPRE il processo e lo introduce nella coda individuata dalla variabile cond \Rightarrow libera il monitor.
- **COND_SIGNAL**: rende attivo un processo in attesa nella coda individuata dalla variabile cond; nel corso della coda vuota non ci sono effetti collaterali.

REAZIONE DEI MONITOR TRAMITE SEMAFORI

Il compilatore invia ad ogni istanza di un monitor:

- Un semolore ^{MUTEX} inizializzato ad 1 per la mutua esclusione delle procedure da monitorare remotico Koso
 - Un semolore URGENT inizializzato a 0 per effettuare la PREEMPTION dei processi.
Deti \rightarrow garantisce ai processi normali di condurreit l'immediato rinvio dopo cond. signal
 - Un contatore URGENTCOUNT inizializzato a 0 per conteggiare in ogni istante ^{processi} Vengono eseguiti
 - Variabile controllata da un semolore CONSEMM inizializzato a 0 ed un contatore ^{processi} condannato inizializzato a 0 per implementare cond. wait e cond. signal

Ogni PROCEDURE ENTRY viene eseguita direttamente dal compilatore tramite un prologo ed un epilogo → PROLOGO wait(mutex); blocco monitor in modo esclusivo

EPICOGO if urgent count > 0

then signal (urgent) \Rightarrow circos procedimento maglie
else signal (normal) \Rightarrow altrimenti libera il monitor

operazioni \Rightarrow cond.wait \rightarrow condcount++;
 if urgentcount > 0
 then signal(~~urgent~~)
 else signal(mutex)
 wait(condsem)
 condcount--;

si tiene traccia delle sospensioni
 se ci sono processi neri per preemption
 negli slot
 altrimenti libera il monitor
 rende il monitor nel semaforo di cond
 il processo non è più nero

cond.signal \Rightarrow urgentcount++;
 if condcount > 0
 signal(~~condsem~~)
 signal(urgent)
 wait(urgent)

si tiene traccia dello present
 se ci sono processi neri
 rimuove il nero
 blocca il processo segnalato

Pmt;
 urgentcount--;

il processo non è più nero

PROBLEMA CHIARATE INNESTATE

Due monitor A e B, procedure entry di A si richiama b_1 e b_1 segue una cond. signal.

La cond. signal viene eseguita da una procedura di b_2 di B che, in tal modo, può interrompere solo da una procedura di A \rightarrow cond. wait libera il monitor B ma monitor A è ancora occupato.

THREAD 1

call A. a_1

A. a_1

a_{11}

a_{12}

a_{1n}

cond. wait(?)

b_{12}

(SOLUZIONI):

ONE BIG CLOCK

\rightarrow T1 detiene il mutex MA per tutto il tempo di accesso ai monitor A e B

THREAD 1

call A. a_1 \rightarrow A. a_1

wait(mA)

a_{11}

call B. b_1 \rightarrow B. b_1

b_{11}

b_{12}

cond(B.wait)()

b_{12}

signal(mB)

Lock(mA)

a_{12}

signal(mA)

SEVERAL SMALL CLOCKS \rightarrow T1 detiene il mutex MA per intervalli più brevi, non ci sono chiamate immediate. Il BIG CLOCK semplifica la vita del programmatore ma meno imposto in modo significativo le prestazioni.

T1

call A. a_1 \rightarrow A. a_1

Lock(mA)

a_{11}

signal(mA)

chiamate immediate. Il BIG CLOCK semplifica la vita del programmatore ma meno imposto in modo significativo le prestazioni.

ma lo stesso impatto in modo significativo le prestazioni.

call B. b_1

B. b_1 :

wait(mB)

b_{11}

cond(B.wait)()

b_{12}

signal(mB)

Lock
mB

call A. a_{12} \rightarrow A. a_{12}

Lock(mA)

a_{12}

signal(mA)

MONITOR IN JAVA

Ad ogni oggetto è associata una variabile di lock, per evitare le multe escezzionali
nella quale viene synchronized → per metodi e per oggetti.

Per metodi il lock viene acquisito all'entrata del metodo synchronize e rilasciato alla uscita. Il lock è ricomune, il non si blocca se già dimessa del lock. In JAVA non c'è garanzia di ordinamento temporale delle richieste d'acquisto del lock.

I metodi che manipolano l'unica variabile di condizione sono:

- WAIT → rilascia il lock anche se non è detenuto in modo esclusivo il controllo passa al thread che ha chiamato
- NOTIFY → risveglia un thread arbitrario tra quelli in attesa. Il thread che risveglia mantiene il lock.
- NOTIFY ALL → mette il mutex su tutti i threads bloccati sulla condizione.

ESEMPIO PRODUTTORE CONSUMATORE

```
public synchronized void deposit(double value) {
    while (count == maxSlots)
        try { wait(); } catch (InterruptedException) {}
    buffer[m] = value;
    m = (m + 1) % slots;
    count++;
    if (count == 1) notify();
}
```

```
public synchronized double deposit() {
```

```
    double value;
    while (count == 0)
        try { wait(); } catch (InterruptedException) {}
    value = buffer[t];
    t = (t + 1) % maxSlots;
    count--;
    if (count == maxSlots - 1) notify();
    return value;
}
```

ESEMPIO MONITOR

• PRODUTTORE - CONSUMATORI

rispetto alle soluzioni con semafori qui non è nemmeno necessario

in posizioni del buffer diverse in nello stesso

1) definizione monitor (variabili e procedenze entry)

type mailbox = monitor {

 l'uso degli oggetti

 Var buffer: array [0..N-1] of messaggio;

 Cont: 0..N;

 non pieno, non vuoto, condition

 procedura entry rend (x: messaggio) {

 begin if cont=N then non-pieno.wait();

 buffer[Cont]:=x;

 Cont:=(Cont+1) mod N;

 non vuoto.signal();

 end

 procedura entry receive (var x: messaggio))

 begin

 if Cont=0 then non-vuoto.wait();

 x:=buffer[Cont];

 Cont:=(Cont+1) mod N;

 non-pieno.signal();

 end

 1) inizializzazione

 begin cont:=0; Testo:=0; Codice:=0;

 esempio intorno Thread

Var a, b: messaggio;

B: mailbox

P:

B.rend(a)

...

C:

B.receive(b)

...

• CESTORE - SCATTORE

Cype lettore-writteri = monitor {

Var num-letteri: integer;
occupato: boolean;
L-letteri: condition;
X-writteri: condition;

procedure entry inizio-letteri

begin

if (occupato or ok-writteri.queue) then ok-letteri.wait();

num-letteri++

ok-letteri.signal();

end

procedure entry fine-letteri

begin

num-letteri--;

if num-letteri = 0 then ok-writteri.signal();

end

procedure entry inizio-writteri

begin

if ((num-letteri < 0 or occupato) then ok-writteri.wait();

occupato = true;

end

procedure entry fine-writteri

begin

occupato = false;

if ok-letteri.queue then ok-letteri.signal() else ok-writteri.signal();

end

FILOSOFIA CENA

Type dining-philosopher = monitor {

Var self: array [0..N-1] of (thinking, hungry, eating)

self: array [0..N-1] condition;

procedure entry pickUp (i: 0..N-1) {

state[i] = hungry;

if state[(i+N-1) mod N] <> eating and state[(i+1) mod N] <> thinking

then state[i] = eating

else: self[i].wait();

}

procedure entry putDown (i: 0..N-1) {

state[i] = eating;

if state[(i+N-1) mod N] == hungry and state[(i+N-2) mod N] <> eating then

{

state[(i+N-1) mod N] = eating;

self[(i+N-1)].signal();

}

if state[(i+1) mod N] hungry and state[(i+2) mod N] <> eating then

{

state[(i+1) mod N] = eating;

self[(i+1) mod N].signal()

}